HW#4: RTOS Analysis

朱致伶, 110550062

***Abstract* — This report focusses on context switching and synchronization with mutex and critical section. Recording the observation of detailed algorithm description and analyze the performance of the above two mechanism respectively.**

I. INTRODUCTION

In this homework, we first take a look at the task control block (TCB) structure type and how the task is created. Secondly, we will have basic knowledge of how task scheduler is initialized and worked. Then, we will have detailed description of the following two mechanisms: context switch, synchronization implemented with mutex and critical section. Finally, we will discuss the overhead of context switch and synchronization. The workspace of this assignment is the same as HW3, however, combined with FreeRTOS. With the ILA as assistance, trace the dataflow of rtos_run.c given by the course in both Aquila_SoC and FreeRTOS.

II. PRE-KNOWLEDGE

For the purpose of obtain more comprehensive understanding of context switch and sychronization mechanism, I would like to introduce the following term in FreeRTOS perspective:

A. Task

In FreeRTOS, tasks are threads with priority, which can be equivalent to the conventional understanding of a process in operating system. While a process requires a Process Control Block (PCB) to save its state, a task also needs a Task Control Block (TCB) to save the current task's state. TCB plays a critical role during context switch, which involves switching between tasks, information about tasks needs to be saved and loaded.

The below paragraphs introduce parameters for context switching and dataflow of how task is created.

The TCB structure information can be found in task.c, named as *tskTaskControlBlock*. TCB includes task information such as the task name, task priority (ranging from 0 to *configMAX_PRIORITIES-1*), and others. Examples such as *xStateListItem* denotes the state of that task which is out of running state (ready, blocked, suspended state), while *xEventListItem* is which used to reference a task from. It is noteworthy that the difference between blocked and suspended state is the initiative. Tasks may fall into blocked state passively, while turn into suspended state actively.

There are some global parameters which are frequently used in the assignment. *xTickCount*, records the number of tick interrupts. *xSchedulerRunning* records whether scheduler is activated. *pxCurrentTCB*, stores the TCB pointer of current running task. *pxReadyTasksLists[configMAX_PRIORITIES]* maintain the ready tasks lists according to their priority level, which shows FreeRTOS implement scheduling with multi-level queue, each priority level has its own queue.

Knowing parameters of TCB allows closer look at how *xTaskCreate* in task.c creates a task. The process of creating a task can be divided to mainly three steps: 1) Check whether the growth direction is forward or backward. Call *pvPortMalloc* and *pvPortMallocStack* to allocate memory space for TCB and stack. 2) Initialize parameters mentioned in previous paragraph of TCB in *prvInitialiseNewTask*. 3) Add the created task into ready list correspond to its priority with *prvAddNewTaskToReadyList,* including initialization of tasklist using *prvInitialiseTaskLists()*, add task to ready list using *prvAddTaskToReadyList* and check whether the current running task has the highest priority of all ready tasks. Noticed that it is necessary to ensure entering critical section when adding new task to ready list, since interrupts shouldn't access the task lists while the lists are being updated. With the above three steps, tasks are created successfully.

B. Scheduler

Before knowing how FreeRTOS implements the scheduler, it is crucial to understand how it chooses the running task with highest priority among several ready tasks.

When adding to the ready list, it is necessary to consider whether the scheduler is operating. If the scheduler is not yet enabled, in other words, *xSchedulerRunning==pdFALSE*, *pxCurrentTCB* will continuously update to the task with a higher priority or the one added later but with the current highest priority. This means that the very first task executed at the beginning will be the one with the highest priority among the tasks created. If they have the same priority, the task created later will take precedence. On the other hand, if the scheduler is already running, it checks whether the priority of the created task is higher than current task. If so, a context switch occurs. Therefore, this explains why, in the provided code, task2 needs to be delayed; it is because task2 is created later and has the same priority as task1. Task2 will be executed first if there is no delay.

It is appropriate to take *vTaskStartScheduler* as an entry point of tracing the scheduler dataflow, since it usually follows after *xTaskCreate* and is called in main.c.

The process of starting a task scheduler can be divided into three sections: 1) Create idle task in either static or dynamic approach. Idle task has the lowest priority, which is created due to the purpose of scheduling. 2) Disable interruption to ensure a tick does not occur before or during the call to *xPortStartScheduler()*. Then, initialize the TCB parameters of idle task related to scheduling, such as setting *xTickCount* to 0, *xSchedulerRunning* to *pdTRUE*. 3) Call *xPortStartScheduler()* to start scheduling.

There are two main function under *xPortStartScheduler()*, which are *vPortSetupTimerInterrupt()* and *xPortStartFirstTask()*. *vPortSetupTimerInterrupt()*, as it named, setup the timer interrupt by two parameters, *mtime* and *mtimecmp*. *A timer interrupt will be triggered whenever mtime>=mtimecmp*. For periodic timer interrupt, set *mtime* to 0 after each interrupt. *xPortStartFirstTask()*, load the address of *freertos_risc_v_trap_handler* to *mtvec* register. *mtvec* is a register that stores the interrupt service routine(ISR) entry point information, which is composed of {BASE[31:2], MODE[1:0]}. Also, interrupts are first enable, and the information stored in *pxCurrentTCB* are loaded to registers, such as *mstatus*, *mepc*, in *xPortStartFirstTask()*.

As we talk about the timer, let's look at how *mtimecmp* is calculated. In *FreeRTOSConfig.h*, the *configCPU_CLOCK_HZ* and *configTICK_RATE_HZ* are defined 41666667 in type uint32_t and 100 in type TickType_t respectively. Since the value of *uxTimerIncrementsForOneTick* in port.c is *configCPU_CLOCK_HZ/configTICK_RATE_HZ*, which is 416666hz. Therefore, as the *mtime* counter in clint.v calculated to 416666, which is the value of *mtimecmp*, the timer interrupt is triggered.

III. Context Switch

This section discusses the process of context switch in FreeRTOS. First, trace the dataflow of context switch in Aquila and FreeRTOS respectively. Divide the dataflow to two parts is not a realistic choice since the code of two parts call each other frequently when running rtos_run.c. However, it is better to understand the algorithmic description of context switch according to my experience of tracing code. Then, analyze the relation between context switching overhead and time quantum.

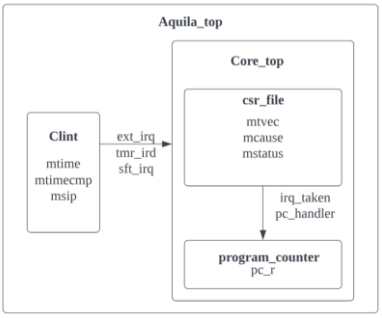
Here is the supplement of the difference between synchronous, asynchronous and external interrupt. The difference between asynchronous and synchronous lies in the former being externally interrupted, which happens at any time, not tied to specific point. While the latter involves voluntarily relinquishing during its execution phase, which is predictable. As for external interrupt, it is often triggered by peripheral devices, which is not handled in Aquila source code.

A. Dataflow in Aquila

The interrupt request signals are triggered in RISC-V Core Local Interrupt (CLINT) Controller. Take clint.v as the starting point. *mtime* and *mtimecmp* controls the timer interrupt request signal. *msip* controls the software interrupt request signal. The output timer and software interrupt signals of clint.v are sent to core_top.v module along with external interrupt signal.

In core_top.v, the three interrupt request signals are first send to csr_file module combined with *irq_enable* signal. In csr_file, it determines whether the interrupt request is taken or not, and the next program counter values if the request is taken with *irq_taken* and *pc_handler* signals. *irq_taken* signals are determined based on the preveliged level(machine, supervisor, user mode), which is stored in *mstatus* register. *irq_taken* signal is triggered only in machine mode. As for *pc_handler*, it is determined by the address of interrupt service routine(ISR) entry point and the cause of interrupt, which are stored in *mtvec* and *mcause* registers respectively. To be noticed that, *mtvec* register is composed of {BASE[31:2], MODE[1:0]}. In direct mode(Mode 0), the ISR address is set to BASE; in vector mode(Mode 1), the ISR address is set to BASE+4*CAUSE. *mtvec* and *mstatus* registers are critical in context switch process since Aquila and FreeRTOS mainly communicate with these registers.

The *irq_taken* and *pc_handler* signals are sent from csr_file to program_counter module. If the interrupt request signal is taken, the address stored in *pc_handler* is given to *pc*_r. Then the processor will find the instruction of *pc_r* in instruction memory and implement the instruction which *pc_r* pointed to.



B. Dataflow in FreeRTOS

Modern CPU has two types of traps: exception(internal) and interrupts(external). The trap handling routines begin its execution in machine mode. Regardless of whether an interrupt or exception occurs, it will enter the FreeRTOS_*riscv*_trap_handler first. Therefore, I take *FreeRTOS*_riscv_trap_handler as the entry point. It is difficult to understand the dataflow with objdump file, so I trace with the portASM.S file instead.

Before performing any operations, it is necessary to first back up the registers of all current tasks. Then, load the current task and its TCB, determine whether it is synchronous or asynchronous. Since this assignment focuses on context switch, we only discuss the asynchronous interrupt part.

After determining it is an asynchronous interrupt, an additional check is made to see if it is an external interrupt issued by a peripheral device. However, external interrupts are not handled in the Aquila source code, so the processing of external interrupts is not discussed in the flowchart. After confirming it is not an external interrupt, the value of *mtime* and *mtimecmp* are obtained. Add *ullNextTime* to the timer increments for one tick.

Then, it switches to ISR stack and jumps to the function *xTaskIncrementTick*, which indicates a tick interrupt has occurred. Updating variables related to the tick is needed in *xTaskIncrementTick*, such as *xTickCount*, *pxTCB* and *xItemValue*. The return value of this function, *xSwitchRequired*, will determine whether a context switch is needed. There are two prerequisites for a context switch: 1) It must be defined as preemptive (non-cooperative) in *FreeRTOSConfig*.h. 2) The current task's priority must be greater than or equal to the priority of the current task. A context switch is needed even if the priority is equal because, in preemptive scheduling, tasks with the equal priority must share the processing time(time slice).

If it determines that a context switch is needed, it goes to *vTaskSwitchContext*. *vTaskSwitchContext* first suspends the scheduler. Secondly, check for stack overflow with

taskCHECK_FOR_STACK_OVERFLOW(). Then, through the two functions, *taskSELECT_HIGHEST_PRIORITY_TASK*() and *traceTASK_SWITCHED_IN*(), we are able to find the next task with highest priority in ready list, and *pxCurrentTCB* holds a pointer to TCB of selected block.

The last step of context switch process eventually goes to *processed_source*, regardless of whether a context switch is performed. In processed_source, *pxCurrentTCB* is loaded into the register, completing the entire tick interrupt process.

C. Context Switch Overhead vs. Time Quantum

A preemptive multi-tasking operating system uses timer interrupt to assign CPU usage from one task to the other. The time slice and context switch are closely related. If a longer time quantum is allocated, it implies a reduction on frequency of context switches, increasing the context switch overhead. Conversely, If the time quantum decreases, the frequency of context switches increases, leading to higher context switch overhead and low efficiency. Table 1 follow the above assumption. As the time quantum increases, the frequency of context switch decreases.

TABLE I. CONTEXT SWITCH FREQUENCY VS. TIME QUANTUM

Time quantum	***5 msec***	***10 msec***	***20 msec***
Frequency	992 times	507 times	248 times
Overhead	585 cycles	512 cycles	434 cycles

The context switching overhead is inversely proportional to the time quantum. To measure the context-switching overhead, I calculated the number of cycles between timer interrupt arrives and a new task starts execution, which are total cycle counts from entering *FreeRTOS_riscv_trap_handler* to leaving processed*_source*.

IV. MUTEX FOR SYNCHORNIZATION

Managing shared resources and synchronizing tasks are undoubtedly important issues in operating systems, which are also key points of this assignment. Before going through the synchronization mechanism, let’s take a deep dive into the concept of mutex and critical section, and the priority inheritance mechanism in FreeRTOS.

In FreeRTOS, mutex can be considered as a type of semaphore. Mutex is used to protect shared resouce, such as memory space and parameters, so that multiple tasks can modify the variable without corrupting each other. Mutex must be locked and unlocked by the same task, which is implemented by taking and giving the semaphore signal. During the time slice, which the mutex is taken but not yet given back, no other tasks except the task unlocked the mutex can access the shared resource between code where mutex are taken and given, until the mutex is given by the exact same task. Critical section are used to protect the UART devices so that different tasks can print concorrent messages properly.

Since mutex in FreeRTOS has a priority inheritance mechanism, so a task 'taking' a mutex type semaphore MUST ALWAYS 'give' the mutex type semaphore back once the mutex it is no longer required.

The priority inheritance mechanism temporarily elevates the priority of a lower-priority task. After completing, the priority is restored to its original priority. Considering a scenario without priority inheritance, if a lower-priority task(t1) acquires shared resource, and a higher-priority task(t2) emerges later, requiring the resource held by t1. A situation may arise where t2 cannot utilize the resource while t1 retains it, leading to a potential deadlock. With priority inheritance, the priority of t1 can be temporarily increased, allowing it to smoothly execute its tasks before releasing the resource. Notice that, the case mentioned above is also the reason that mutex cannot be used from within interrupt service routines.

With the knowledge of mutex, critical section and priority inheritance mechanism, the below section shows the process of creating, taking and giving mutex:

A. Mutex Creation

In rtos_run.c, mutex are created with *xSemaphoreCreateMutex()* and *prvInitialiseMutex().* The former calls the *xQueueGenericCreate* function that truly create mutex in queue structure. *xQueueGenericCreate* allocates space for the new queue using *pvPortMalloc* and initializes it using *prvInitialiseNewQueue*. The latter overwrite part of the queue structure member due to the priority inheritance mechanism, calling *xQueueGenericSend* which truly overwrite the members.

The queue structure information is stored in queue.c of structure type *QueueDefinition*. Queue structure includes important members, such as *pcHead* (pointing to the head start of the queue), *uxLength*(records the maximum length of queue), *uxItemSize*(records the item sizes store in the queue), *uxMessagesWaiting*(records the current number of items in the queue), *xTasksWaitingToSend* and *xTasksWaitingToReceive* (stores the task read and write from queue causing the mutex block) and others.

It's worth noting that *xQueueCreateMutex* specifies *uxLength* as 1 and *uxItemSize* as 0 when calling *xQueueGenericCreate*. The reason for *uxItemSize* being 0 is that a mutex, unlike a regular queue, doesn't need to store item data; it only relies on the queue's data structure itself.

Although *xSemaphoreCreateMutex()* set all the queue structure members correctly for a generic queue, but what we need is a mutex type semaphore. In *prvInitialiseMutex(),* part of queue structure members are overwrote, since it is needed to be set differently in particular the information required for priority inheritance. The *uxQueueType* is set to *queueQUEUE_IS_MUTEX*. Then, the *xQueueGenericSend* is called, which I considered as one of the most complicated parts in FreeRTOS.

In *xQueueGenericSend*, check whether the scheduler is suspended first. If the scheduler is suspended, the mutex can not be blocked. The process of *xQueueGenericSend* can be divided into two parts: operations in critical section, operations after exiting critical section. Noticed that *xQueueGenericSend* is also implemented when giving a mutex.

During the critical section, first check whether there are empty spaces in the current queue. There are two cases which satisfy the condition: 1) As the value of *uxMessagesWaiting* is smaller than *uxLength*, which is quite intuitive. 2) As xCopyPosition == *queueOVERWRITE*, since overwrite new item on old one doesn't require empty space. Secondly, implement a deep copy on the item going to push into the queue with *prvCopyDataToQueue*. Also, the task priority is disinherited, since mutex is no longer being held in *prvCopyDataToQueue*. Thirdly, determining whether the queue is a member of queue set. If true, call *prvNotifyQueueSetContainer* to inform the queue set that queue has been modified. The return value of *prvNotifyQueueSetContainer* decides if context switch is required or not, and implement the preemption. Moreover, if the queue list isn't empty, which indicates the unblocked task may have a priority higher than our own, so yield immediately. On the contrary, if the queue does not belong to any queue set or is not configured with a queue set in *FreeRTOSConfig*.h, it then checks whether any task is waiting for data inside the queue. If there is, it unblocks the task and assesses whether its priority is higher than the current task. If so, a context switch occurs; otherwise, it is a non-operation (NOP).

On the other hand, if the queue is already full and cannot be overwritten, it needs to determine how long it is willing to wait, as specified by *xTicksToWait*. If it is 0, the function exits the critical section and immediately returns. If it is nonzero, it sets the timeout period using *vTaskInternalSetTimeOutState* and *xTiimeOut*. Entering this state implies that the current task is in a blocked state due to the queue being full.

After exiting the critical section, the task can read and write to the queue. Subsequently, it invokes *vTaskSuspendAll*() and *prvLockQueue*. The former temporarily halts scheduling to prevent task switches, while the latter secures the queue to prevent interference from interrupts that might trigger a context switch. Lastly, it verifies whether the current time has exceeded the specified timeout duration. If it has, the queue is unlocked, scheduling is restored, and the function returns. If not, it assesses the available space in the current queue. If the queue is full, *vTaskPlaceOnEventList* is called to append the current task's event list to the delay list.

B. Mutex Taken

In rtos_run.c, mutex are taken with *xSemaphoreTake*, which calls *xQueueSemaphoreTake* that truly takes the mutex, turning it into block mode.

The mutex taken process can also be divided into two parts: operations in critical section, operations after exiting critical section. First, check the queue is in mutex type with item size equal to 0. Semaphores are queues with item size equals 0, and where the number of messages in the queue is the semaphore's count value. During the critical section, if there is existing data in current queue, copy the information required to implement priority inheritance which will be used later. Then, check to see if other tasks are blocked waiting to give the semaphore, and if so, unblock the highest priority such task. The process may requires context switch with *queueYIELD_IF_USING_PREEMPTION()*. If there is no existing item in queue, it determines how long it is willing to wait with xTicksToWait. If it is 0, the function exits the critical section and immediately returns. If it is nonzero, it sets the timeout period using *vTaskInternalSetTimeOutState* and *xTimeOut*. The semaphore count is 0 and a block time is specified so configure the timeout structure ready to block. Finally, it is allowed to exit the critical section.

After exiting the critical section, operations of taking a mutex are similar to operations of creating mutex. The only difference is that the *xSemaphoreTaken* has to implement priority inheritance and disinheritance mechanism. If task check for timeout shows the task isn't expired, do the prioirty inheritance; otherwise, implement the priority disinheritance. In *xTaskPriorityInherit*, if the holder of the mutex has a priority below the priority of the task attempting to obtain the mutex then it will temporarily inherit the priority of the task attempting to obtain the mutex. *vTaskPriorityDisinheritAfterTimeout* determines the priority to which the priority of the task that holds the mutex should be set. The set priority of holder task must be greater than holding task's base priority and the priority of the highest priority task that is waiting to obtain the mutex.

C. Mutex Given

In rtos_run.c, mutex are given with *xSemaphoreGive*, which calls *xQueueGenericSend* that truly given the mutex, turning it into unblock mode. As for the detailed implementation of *xQueueGenericSend* has already been introduced in the previous section of mutex creation, we will skip the description here. The synchronization overhead of taking and giving mutex I calculated are totcal cycle counts in *xQueueSemaphoreTake* and *xQueueGenericSend*.

TABLE II. OVERHEAD OF TAKE/GIVE A MUTEX

	Take a Mutex	***Give a Mutex***
Synchronization Overhead	209 cycles	278 cycles

D. Critical Section

In rtos_run.c, the critical section is somewhat different from mutex and semaphore. Entering a critical section indicates the intent to perform an uninterrupted operation. The implementation of critical section is much simpler compared to mutex and semaphore, and it ultimately traces back to *vTaskEnterCritical* and *vTaskExitCritical*. This function is responsible for disabling interruptions and maintaining a count of the nesting level. The purpose of the nesting level becomes apparent when exiting the nested critical sections with *vTaskExitCritical*. Interrupts are only re-enabled when returning to the outermost critical section level.The synchronization overhead of entering and leaving critical section I calculated are totcal cycle counts in *vTaskEnterCritical* and *vTaskExitCritical* .

TABLE III. OVERHEAD OF ENTER/LEAVE CRITICAL SECTION

	Enter Critical Section	***Leave Critical Section***
Synchronization Overhead	36 cycles	51 cycles